filosof hanya dapat berstatus makan (*eating*) jika tidak ada tetangganya yang sedang makan juga. Tetangga seorang filosof didefinisikan ole LEFT dan RIGHT.

Dengan kata lain, jika i = 2, maka tetangga kirinya (LEFT) = 1 dan tetangga kanannya (RIGHT) = 3. Program ini menggunakan sebuah array dari semaphore yang lapar (hungry) dapat ditahan jika garpu kiri atau kanannya sedang dipakai tetangganya. Catatan bahwa masing-masing proses menjalankan prosedur filosof sebagai kode utama, tetapi prosedur yang lain seperti take-forks, dan test adalah prosedur biasa dan bukan proses-proses yang terpisah.

3.1.6. Monitors

Solusi sinkronisasi ini dikemukakan oleh Hoare pada tahun 1974. Monitor adalah kumpulan prosedur, variabel dan struktur data di satu modul atau paket khusus. Proses dapat memanggil prosedur-prosedur kapan pun diinginkan. Tapi proses tak dapat mengakses struktur data internal dalam monitor secara langsung. Hanya lewat prosedur-prosedur yang dideklarasikan minitor untuk mengakses struktur internal.

Properti-properti monitor adalah sebagai berikut:

i. Variabel-variabel data lokal, hanya dapat diakses oleh prosedur-prosedur dala monitor dan tidak oleh prosedur di luar monitor.

ii. Hanya satu proses yang dapat aktif di monitor pada satu saat. Kompilator harus mengimplementasi ini(mutual exclusion).

iii. Terdapat cara agar proses yang tidak dapat berlangsung di-blocked. Menambahkan variabel-variabel kondisi, dengan dua operasi, yaitu Wait dan Signal.

iv. ait: Ketika prosedur monitor tidak dapat berlanjut (misal producer menemui buffer penuh) menyebabkan proses pemanggil diblocked dan mengizinkan proses lain masuk monitor.

v. Signal: Proses membangunkan partner-nya yang sedang diblocked dengan signal pada variabel kondisi yang sedang ditunggu partnernya.

vi. Versi Hoare: Setelah signal, membangunkan proses baru agar berjalan dan menunda proses lain. vii. Versi Brinch Hansen: Setelah melakukan signal, proses segera keluar dari monitor.

Dengan memaksakan disiplin hanya satu proses pada satu saat yang berjalan pada monitor, monitor menyediakan fasilitas mutual exclusion. Variabel-variabel data dalam monitor hanya dapat diakses oleh satu proses pada satu saat. Struktur data bersama dapat dilindungi dengan menempatkannya dalam monitor. Jika data pada monitor merepresentasikan sumber daya, maka monitor menyediakan fasilitas mutual exclusion dalam mengakses sumber daya itu.

3.2. *Deadlock*

Pada pembahasan di atas telah dikenal suatu istilah yang populer pada bagian *semaphores*, yaitu *deadlock*. Secara sederhana *deadlock* dapat terjadi dan menjadi hal yang merugikan, jika pada suatu saat ada suatu proses yang memakai sumber daya dan ada proses lain yang menunggunya. Bagaimanakah *deadlock* itu yang sebenarnya? Bagaimanakah cara penanggulangannya?

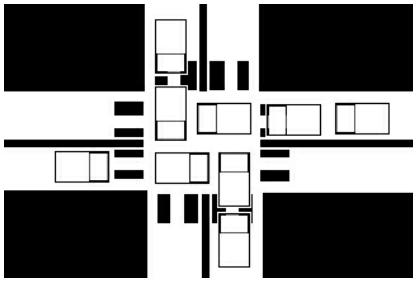
3.2.1. Latar Belakang

Misalkan pada suatu komputer terdapat dua buah program, sebuah *tape drive* dan sebuah *printer*. Program A mengontrol *tape* drive, sementara program B mengontrol *printer*. Setelah beberapa saat, program A meminta *printer*, tapi *printer* masih digunakan. Berikutnya, B meminta *tape drive*, sedangkan A masih mengontrol *tape drive*. Dua program tersebut memegang kontrol terhadap sumber daya yang dibutuhkan oleh program yang lain. Tidak ada yang dapat melanjutkan proses masing-masing sampai program yang lain memberikan sumber dayanya, tetapi tidak ada yang mengalah. Kondisi inilah yang disebut *Deadlock* atau pada beberapa buku disebut *Deadly Embrace Deadlock* yang mungkin dapat terjadi pada suatu proses disebabkan proses itu menunggu suatu kejadian tertentu yang tidak akan pernah terjadi. Dua atau lebih proses dikatakan berada dalam kondisi *deadlock*, bila setiap proses yang ada menunggu suatu kejadian yang hanya dapat dilakukan oleh proses lain dalam himpunan tersebut.

Terdapat kaitan antara *overhead* dari mekanisme koreksi dan manfaat dari koreksi *deadlock* itu sendiri. Pada beberapa kasus, *overhead* atau ongkos yang harus dibayar untuk membuat sistem bebas deadlock menjadi hal yang terlalu mahal dibandingkan jika mengabaikannya. Sementara pada kasus lain, seperti pada *real-time process control*, mengizinkan *deadlock* akan membuat sistem menjadi kacau dan membuat sistem tersebut tidak berguna.

Contoh berikut ini terjadi pada sebuah persimpangan jalan. Beberapa hal yang dapat membuat *deadlock* pada suatu persimpangan, yaitu:

- Terdapat satu jalur pada jalan.
- Mobil digambarkan sebagai proses yang sedang menuju sumber daya.
- Untuk mengatasinya beberapa mobil harus *preempt* (mundur).
- Sangat memungkinkan untuk terjadinya *starvation* (kondisi proses tak akan mendapatkan sumber daya).

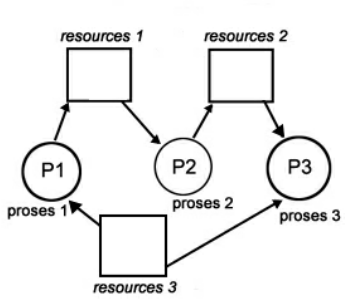


Gambar 3-13. Persimpangan.

3.2.2. *Resources-Allocation Graph*

Sebuah cara visual (matematika) untuk menentukan apakah ada *deadlock*, atau kemungkinan terjadinya. G = (V, E) Graf berisi *node and edge*. *Node* V terdiri dari proses-proses = {P1, P2, P3, ...} dan jenis *resource*. {R1, R2, ...} *Edge* E adalah (Pi, Rj) atau (Ri, Pj) Sebuah panah dari *process* ke *resource* menandakan proses meminta *resource*.

Sebuah panah dari *resource* ke *process* menunjukkan sebuah *instance* dari *resource* telah dtempatkan ke proses. *Process* adalah lingkaran, *resource* adalah kotak; titik-titik merepresentasikan jumlah *instance* dari *resource* Dalam tipe. Meminta poin-poin ke kotak, perintah datang dari titik.

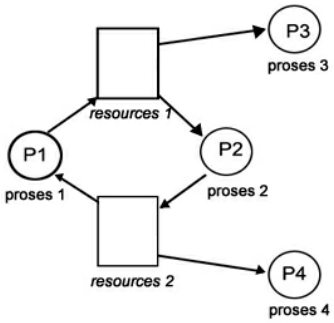


Gambar 3-14. Graph.

Jika graf tidak berisi lingkaran, maka tidak ada proses yang *deadlock*.

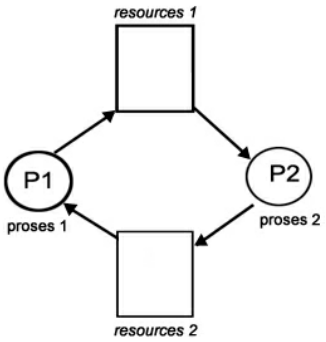
Jika membentuk lingkaran, maka:

i. jika tipe *resource* memiliki banyak *instance*, maka *deadlock* DAPAT ada.



Gambar 3-15. Non Deadlock.

ii. jika setiap tipe *resource* mempunyai satu *instance*, maka *deadlock* telah terjadi.



Gambar 3-16. Deadlock.

3.2.3. Model Sistem

Menurut *Coffman* dalam bukunya *"Operating System"* menyebutkan empat syarat bagi terjadinya *deadlock*, yaitu:

i. Mutual Exclusion
Suatu kondisi dimana setiap sumber daya diberikan tepat pada satu proses pada suatu waktu.

ii. Hold and Wait
Kondisi yang menyatakan proses-proses yang sedang memakai suatu sumber daya dapat meminta sumber daya yang lain.

iii. Non-pre-emptive
Kondisi dimana suatu sumber daya yang sedang berada pada suatu proses tidak dapat diambil secara paksa dari proses tersebut,sampai proses itu melepaskannya.

iv. Circular Wait
Kondisi yang menyatakan bahwa adanya rantai saling meminta sumber daya yang dimiliki oleh suatu proses oleh proses lainnya.

3.2.4. Strategi menghadapi *Deadlock*

Strategi untuk menghadapi *deadlock* dapat dibagi menjadi tiga pendekatan, yaitu:

i. Mengabaikan adanya *deadlock*.

ii. Memastikan bahwa *deadlock* tidak akan pernah ada, baik dengan metode. Pencegahan, dengan mencegah empat kondisi *deadlock* agar tidak akan pernah terjadi. Metode Menghindari *deadlock*, yaitu mengizinkan empat kondisi *deadlock*, tetapi menghentikan setiap proses yang kemungkinan mencapai *deadlock*.

iii. Membiarkan *deadlock* untuk terjadi, pendekatan ini membutuhkan dua metode yang saling mendukung, yaitu:

• Pendeteksian *deadlock*, untuk mengidentifikasi ketika *deadlock* terjadi.
• Pemulihan *deadlock*, mengembalikan kembali sumber daya yang dibutuhkan pada proses yang memintanya.

Dari penjabaran pendekatan diatas, terdapat empat metode untuk mengatasi *deadlock* yang akan terjadi,
yaitu:

3.2.4.1. Strategi *Ostrich*

Pendekatan yang paling sederhana adalah dengan menggunakan strategi burung unta: masukkan kepala dalam pasir dan seolah-olah tidak pernah ada masalah sama sekali. Beragam pendapat muncul berkaitan dengan strategi ini. Menurut para ahli Matematika, cara ini sama sekali tidak dapat diterima dan semua keadaan *deadlock* harus ditangani. Sementara menurut para ahli Teknik, jika komputer lebih sering mengalami kerusakkan disebabkan oleh kegagalan *hardware*, *error* pada kompilator atau *bugs* pada sistem operasi. Maka ongkos yang dibayar untuk melakukan penanganan

deadlock sangatlah besar dan lebih baik mengabaikan keadaan *deadlock* tersebut. Metode ini diterapkan pada sistem operasi UNIX dan MINIX.

3.2.5. Mencegah *Deadlock*

Metode ini merupakan metode yang paling sering digunakan. Metode Pencegahan dianggap sebagai solusi yang bersih dipandang dari sudut tercegahnya *deadlock*. Tetapi pencgahan akan mengakibatkan kinerja utilisasi sumber daya yang buruk.

Metode pencegahan menggunakan pendekatan dengan cara meniadakan empat syarat yang dapatmenyebabkan deadlock terjadi pada saat eksekusi Coffman (1971).

Syarat pertama yang akan dapat ditiadakan adalah *Mutual Exclusion*, jika tidak ada sumber daya yang secara khusus diperuntukkan bagi suatu proses maka tidak akan pernah terjadi *deadlock*. Namun jika membiarkan ada dua atau lebih proses mengakses sebuah sumber daya yang sama akan menyebabkan *chaos*. Langkah yang digunakan adalah dengan *spooling* sumber daya, yaitu dengan mengantrikan *job-job* pada antrian dan akan dilayani satu-satu.

Beberapa masalah yang mungkin terjadi adalah:

i. Tidak semua dapat di-*spool*, tabel proses sendiri tidak mungkin untuk di-*spool*
ii. Kompetisi pada ruang disk untuk *spooling* sendiri dapat mengarah pada *deadlock*

Hal inilah yang menyebabkan mengapa syarat pertama tidak dapat ditiadakan, jadi *mutual exclusion* benar-benar tidak dapat dihilangkan.

Cara kedua dengan meniadakan kondisi *hold and wait* terlihat lebih menjanjikan. Jika suatu proses yang sedang menggunakan sumber daya dapat dicegah agar tidak dapat menunggu sumber daya yang lain, maka *deadlock* dapat dicegah. Langkah yang digunakan adalah dengan membuat proses agar meminta sumber daya yang mereka butuhkan pada awal proses sehingga dapat dialokasikan sumber daya yang dibutuhkan. Namun jika terdapat sumber daya yang sedang terpakai maka proses tersebut tidak dapat memulai prosesnya.

Masalah yang mungkin terjadi:

i. Sulitnya mengetahui berapa sumber daya yang dibutuhkan pada awal proses

ii. Tidak optimalnya pengunaan sumber daya jika ada sumber daya yang digunakan hanya beberapa waktu dan tidak digunakan tapi tetap dimiliki oleh suatu proses yang telah memintanya dari awal.

Meniadakan syarat ketiga *non preemptive* ternyata tidak lebih menjanjikan dari meniadakan syarat kedua, karena dengan meniadakan syarat ketiga maka suatu proses dapat dihentikan ditengah jalan. Hal ini tidak dimungkinkan karena hasil dari suatu proses yang dihentikan menjadi tidak baik.

Cara terakhir adalah dengan meniadakan syarat keempat *circular wait*. Terdapat dua pendekatan, yaitu:

i. Mengatur agar setiap proses hanya dapat menggunakan sebuah sumber daya pada suatu waktu, jika menginginkan sumber daya lain maka sumber daya yang dimiliki harus dilepas.
ii. Membuat penomoran pada proses-proses yang mengakses sumber daya. Suatu proses dimungkinkan untuk dapat meminta sumber daya kapan pun, tetapi permintaannya harus dibuat terurut.

Masalah yang mungkin terjadi dengan mengatur bahwa setiap proses hanya dapat memiliki satu proses adalah bahwa tidak semua proses hanya membutuhkan satu sumber daya, untuk suatu proses yang kompleks dibutuhkan banyak sumber daya pada saat yang bersamaan. Sedangkan dengan penomoran masalah yang dihadapi adalah tidak terdapatnya suatu penomoran yang dapat memuaskan semua pihak.

Secara ringkas pendekatan yang digunakan pada metode pencegahan deadlock dan masalah-masalah yang menghambatnya, terangkum dalam tabel dibawah ini.

Tabel 3-1. Tabel Deadlock

Syarat	Langkah	Kelemahan
Mutual Exclusion	*Spooling* sumber daya	Dapat menyebabkan *chaos*
Hold and Wait	Meminta sumber daya di awal	Sulit memperkirakan di awal dan tidak optimal
No Pre-emptive	Mengambil sumber daya di tengah proses	Hasil proses tidak akan baik
Circular Wait	Penomoran permintaan sumber daya	Tidak ada penomoran yang memuaskan semua pihak

3.2.6. Menghindari *Deadlock*

Pendekatan metode ini adalah dengan hanya memberi kesempatan ke permintaan sumber daya yang tidak mungkin akan menyebabkan *deadlock*. Metode ini memeriksa dampak pemberian akses pada suatu proses, jika pemberian akses tidak mungkin menuju kepada *deadlock*, maka sumber daya akan diberikan pada proses yang meminta. Jika tidak aman, proses yang meminta akan di-*suspend* sampai suatu waktu permintaannya aman untuk diberikan. Kondisi ini terjadi ketika setelah sumber daya yang sebelumnya dipegang oleh proses lain telah dilepaskan.

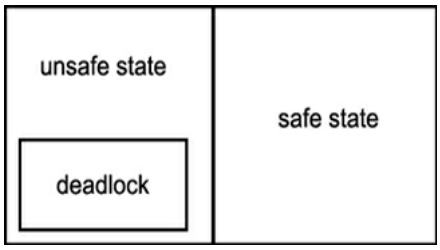
Kondisi aman yang dimaksudkan selanjutnya disebut sebagai *safe-state*, sedangkan keadaan yang tidak memungkinkan untuk diberikan sumber daya yang diminta disebut *unsafe-state*.

3.2.6.1. Kondisi Aman (*Safe state*)

Suatu keadaan dapat dinyatakan sebagai *safe state* jika tidak terjadi *deadlock* dan terdapat cara untuk memenuhi semua permintaan sumber daya yang ditunda tanpa menghasilkan *deadlock*. Dengan cara mengikuti urutan tertentu.

3.2.6.2. Kondisi Tak Aman (*Unsafe state*)

Suatu *state* dinyatakan sebagai *state* tak selamat (*unsafe state*) jika tidak terdapat cara untuk memenuhi semua permintaaan yang saat ini ditunda dengan menjalankan proses-proses dengan suatu urutan.



Gambar 3-17. Safe.

3.2.7. Algoritma *Bankir*

Algoritma penjadualan ini diungkapkan oleh Dijkstra (1965) lebih dikenal dengan nama Algoritma Bankir. Model ini menggunakan suatu kota kecil sebagai percontohan dengan suatu bank sebagai sistem operasi, pinjaman sebagai sumber daya dan peminjam sebagai proses yang membutuhkan sumber daya.

Deadlock akan terjadi apabila terdapat seorang peminjam yang belum mengembalikan uangnya dan ingin meminjam kembali, padahal uang yang belum dikembalikan tadi dibutuhkan oleh peminjam lain yang juga belum mengembalikan uang pinjamannya.
Beberapa kelemahan algoritma Bankir Tanenbaum (1992), Stallings (1995) dan Deitel (1990) adalah sebagai berikut:

i. Sulit untuk mengetahui seluruh sumber daya yang dibutuhkan proses pada awal eksekusi.

ii. Jumlah proses yang tidak tetap dan berubah-ubah.

iii. Sumber daya yang tadinya tersedia dapat saja menjadi tidak tersedia kembali.

iv. Proses-proses yang dieksekusi haruslah tidak dibatasi oleh kebutuhan sinkronisasi antar proses.

v. Algoritma ini menghendaki memberikan semua permintaan selama waktu yang berhingga.

3.2.8. Mendeteksi *Deadlock* dan Memulihkan *Deadlock*

Metode ini mengunakan pendekatan dengan teknik untuk menentukan apakah *deadlock* sedang terjadi serta proses-proses dan sumber daya yang terlibat dalam deadlock tersebut. Setelah kondisi *deadlock* dapat dideteksi, maka langkah pemulihan dari kondisi *deadlock* dapat segera dilakukan. Langkah pemulihan tersebut adalah dengan memperoleh sumber daya yang diperlukan oleh proses-proses yang membutuhkannya. Beberapa cara digunakan untuk mendapatkan sumber daya yang diperlukan, yaitu dengan terminasi proses dan *pre-emption* (mundur) suatu proses. Metode ini banyak digunakan pada komputer *mainframe* berukuran besar.

3.2.8.1. Terminasi Proses

Metode ini akan menghapus proses-proses yang terlibat pada kondisi *deadlock* dengan mengacu pada beberapa syarat. Beberapa syarat yang termasuk dalam metode ini adalah, sebagai berikut:

- Menghapus semua proses yang terlibat dalam kondisi *deadlock* (solusi ini terlalu mahal).

- Menghapus satu persatu proses yang terlibat, sampai kondisi *deadlock* dapat diatasi (memakan banyak waktu).

- Menghapus proses berdasarkan prioritas, waktu eksekusi, waktu untuk selesai, dan kedalaman dari *rollback*.

3.2.8.2. *Resources Preemption*

Metode ini lebih menekankan kepada bagaimana menghambat suatu proses dan sumber daya, agar tidak terjebak pada *unsafe condition*.

Beberapa langkahnya, yaitu:

• Pilih salah satu - proses dan sumber daya yang akan di-*preempt*.
• *Rollback* ke *safe state* yang sebelumnya telah terjadi.
• Mencegah suatu proses agar tidak terjebak pada *starvation* karena metode ini.

3.3. Kesimpulan

Untuk mengatasi problem critical section dapat digunakan berbagai solusi software. Namun masalah yang akan timbul dengan solusi software adalah solusi software tidak mampu menangani masalah yang lebih berat dari critical section. Tetapi Semaphores mampu menanganinya, terlebih jika hardware yang digunakan mendukung maka akan memudahkan dalam menghadapi problem sinkronisasi.

Berbagai contoh klasik problem sinkronisasi berguna untuk mengecek setiap skema baru sinkronisasi. Monitor termasuk ke dalam level tertinggi mekanisme sinkronisasi yang berguna untuk mengkoordinir aktivitas dari banyak thread ketika mengakses data melalui pernyataan yang telah disinkronisasi

Kondisi *deadlock* akan dapat terjadi jika terdapat dua atau lebih proses yang akan mengakses sumber daya yang sedang dipakai oleh proses yang lainnya. Pendekatan untuk mengatasi *deadlock* dipakai tiga buah pendekatan, yaitu:

• Memastikan bahwa tidak pernah dicapai kondisi *deadlock*
• Membiarkan *deadlock* untuk terjadi dan memulihkannya
• Mengabaikan apa pun *deadlock* yang terjadi

Dari ketiga pendekatan diatas, dapat diturunkan menjadi empat buah metode untuk mengatasi *deadlock*, yaitu:

• Pencegahan *deadlock*
• Menghindari *deadlock*
• Mendeteksi *deadlock*
• Pemulihan *deadlock*

Namun pada sebagian besar Sistem Operasi dewasa ini mereka lebih condong menggunakan pendekatan untuk mengabaikan semua *deadlock* yang terjadi Silberschatz

(1994) merumuskan sebuah strategi penanggulangan deadlock terpadu yang dapat disesuaikan dengan kondisi dan situasi yang berbeda, strateginya sendiri berbunyi:

1. Kelompokkan sumber daya kedalam kelas yang berbeda

2. Gunakan strategi pengurutan linear untuk mencegah kondisi *circular wait*yang nantinya akan mencegah deadlock diantara kelas sumber daya

3. Gunakan algoritma yang paling cocok untuk suatu kelas sumber daya yang berbeda satu dengan yang lain

3.4. Latihan

1. Proses dapat meminta berbagai kombinasi dari sumber daya dibawah ini: *CDROM*, *soundcard* dan *floppy*. Jelaskan tiga macam pencegahan deadlock skema yang meniadakan:
   - *Hold and Wait*
   - *Circular Wait*
   - *No Preemption*

2. Diasumsikan proses P0 memegang sumber daya R2 dan R3, meminta sumber daya R4; P1 menggunakan R4 dan meminta R1; P2 menggunakan R1 dan meminta R3 . Gambarkan *Wait-for Graph*. Apakah sistem terjebak dalam *deadlock*? Jika ya, tunjukkan proses mana yang menyebabkan *deadlock*. Jika tidak, tunjukkan urutan proses untuk selesai.

3. User x telah menggunakan 7 printer dan harus menggunakan 10 printer. User y telah menggunakan 1 printer dan akan memerlukan paling banyak 4 printer. User z telah menggunakan 2 printer dan akan menggunakan paling banyak 4 printer. Setiap user pada saat ini meminta 1 printer. Kepada siapakah OS akan memberikan grant printer tersebut dan tunjukkan "safe sequence" yang ada sehingga tidak terjadi deadlock.

4. Pernyataan manakah yang benar mengenai deadlock:
   i. Pencegahan deadlock lebih sulit dilakukan (implementasi) daripada menghindari deadlock.
   ii. Deteksi deadlock dipilih karena utilisasi dari resources dapat lebih optimal.
   iii. Salah satu prasyarat untuk melakukan deteksi deadlock adalah: hold and wait.
   iv. Algoritma Banker's (Djikstra) tidak dapat menghindari terjadinya deadlock.
   v. Suatu sistem jika berada dalam keadaan tidak aman: "unsafe", berarti telah terjadi deadlock.

5. User 1 sedang menggunakan x printers dan memerlukan total n printers. Kondisi umum adalah: y < -12, n < -12, x < -y, m < -n. State ini safe jika dan hanya jika:

   i. x+n<-12 dan y+m<-12 dan x+m<-12
   ii. x+n<-12 dan y+m<12 dan x+m<-12
   iii. x+n<-12 atau(or) y+m<-12 dan x+m<-12
   iv. x+m<-12
   v. Semua statement diatas menjamin: safe state

3.5. Rujukan

3.5.1. Rujukan Sinkronisasi

1. Hariyanto, B.,"*Sistem Operasi*", Bandung: Informatika, Desember 1997
2. Silberschatz, A., Gagne, G. dan Galvin, P., "*Applied Operating System Concept*", John Wiley and Sons Inc., 2000
3. Tanenbaum, Andrew S., "*Modern Operating Systems*", Englewood Cliffs, New Jersey: Prentice-Hall Inc., 1992

3.5.2. Rujukan *Deadlock*

1. Coffman, E.G., Jr., M.J. Elphick dan A. Shoshani, "*System Deadlocks*", Computing surveys, Vol.3, No.2, June 1971
2. Deitel, H.M., "*Operating Systems*", 2nd Edition, Massachusetts: Addison-WesleyPublishing Company, 1990
3. Hariyanto, B.,"Sistem Operasi", Bandung: Informatika, Desember 1997
4. Havender, J.W., "*Avoiding Deadlock in Multitasking Systems*", IBM Systems Journal, Vol.7, No.2, 1968. *97*
5. Silberschatz, A., Gagne, G. dan Galvin, P., "*Applied Operating System Concept*", John Wiley and Sons Inc., 2000
6. Tanenbaum, Andrew S., "*Modern Operating Systems*", Englewood Cliffs, New Jersey: Prentice-Hall Inc., 1992